

Solución 1P

22 de junio de 2023

Update 30/06: En el ejercicio 1 ahora dice “Suponiendo que *se_puede_armar_rima* devuelve 0 en caso afirmativo y 1 en caso contrario.”, estaba al revés.

Ejercicio 1

a) $f_c : \mathbb{N} \rightarrow \mathbb{N}$,

$f(i)$ = “mínimo *cringe* alcanzable para el sufijo de palabras A_i, \dots, A_n con un cantante de talento c ”

Empezamos por observar que como $f(i)$ representa la solución para el sufijo de palabras que arranca en i y termina en n , el primer llamado va a ser con $f(1)$ (si indexamos a las palabras en 1) y el caso base debería estar alrededor de $f(n)$ o $f(n+1)$. Además, $f(i)$ se va a definir en términos de $f(j)$ con $j > i$.

Ahora pensemos qué vamos a hacer en cada paso. Es decir, qué vamos a agregar a nuestra solución parcial entre un llamado recursivo y el siguiente. Si agregásemos un verso, nos tendríamos que acordar en qué termina para decidir cuánto nos cuesta el siguiente verso que elijamos. Como la función sugerida solo tiene a i como parámetro, no podemos hacer esto.

Como cada par de versos tiene que rimar entre sí (el primero con el segundo, el tercero con el cuarto, etc.) en forma independiente al resto, construyamos y agreguemos un par de versos en cada paso. Con este esquema no tendremos que “recordar” nada. Para esto, es importante considerar todas las formas posibles de agregar dos versos. Si suponemos que vamos a usar las palabras A_i, \dots, A_j para el par de versos, la función *se_puede_armar_rima* nos dice si hay alguna forma de separar esas palabras en dos versos rimando y respetando el talento del cantante. Además, usando el recurso siempre podemos hacer rimar un conjunto de palabras en dos versos (siempre y cuando haya al menos dos palabras y no más de $2c$). Luego, solo hace falta analizar todos los j posibles llamando a la función que nos dan para cada uno. La función recursiva quedaría así:

$$f(i) = \begin{cases} \infty, & \text{si } i > n + 1 \\ \infty, & \text{si } i = n \\ 0, & \text{si } i = n + 1 \\ \min\{se_puede_armar_rima(i, j) + f(j + 1) : i + 1 \leq j < i + 2c\}, & \text{cc.} \end{cases}$$

Suponiendo que *se_puede_armar_rima* devuelve 0 en caso afirmativo y 1 en caso contrario. Notar que las palabras están numeradas de 1 a n y los intervalos de palabras incluyen a ambos extremos. Cuando $i = n$ devolvemos ∞ porque no es posible hacer rimar una palabra que quedó sola.

b) Tenemos que ver que la cantidad de llamados recursivos es mayor asintóticamente que la cantidad de subproblemas.

La cantidad de subproblemas es $O(n)$ porque i es el único parámetro de f y $1 \leq i \leq n$ (podemos ignorar los subproblemas con $i > n$ porque se computan en $O(1)$, si quisiéramos podríamos directamente no llamar a f en estos casos y el costo sería el mismo).

Para la cantidad de llamados recursivos, la idea es que claramente son muchos más que n . Por ejemplo, si tomásemos $i < j \leq n$ en vez de $i + 1 \leq j < i + 2c$, serían tantos como los subconjuntos de un conjunto de $n + 1$ elementos¹, pero por la restricción son un poco menos.

Formalmente, esta última idea no nos sirve porque tenemos que acotar por debajo a los llamados (notación Ω). Otra idea sería decir que en cada paso tenemos al menos dos alternativas, intentando justificar que son al menos 2^n llamados. Esto no es completamente cierto porque no siempre hacemos n pasos (podríamos ir avanzando con versos muy grandes). Sin embargo, sí es cierto que hacemos al menos $\frac{n}{2c}$ pasos porque a lo sumo avanzamos $2c$ en cada uno. Luego la cantidad de llamados recursivos es al menos $\Omega(2^{\frac{n}{2c}})$. Como para $2c \ll n$ (ej. $c = 3$) esto es exponencial en n , y claramente $2c \ll n$ es un conjunto de entradas posibles, vale la propiedad de superposición de problemas (por lo menos en esas instancias).

c) Un algoritmo *top-down* para calcular $f_c(i)$ consistiría en implementar $f_c(i)$ recursivamente, usando un vector de longitud n como estructura de memoización. Luego, el costo asintótico sería a lo sumo el producto entre el tamaño de esta memoria (n) y el costo de cada llamado recursivo. Cada llamado recursivo es $O(c^2)$ porque hacemos $O(2c) = O(c)$ llamados a *se_puede_armar_rima*, que cuesta $O(c)$. Luego, la complejidad es $O(n * c^2)$.

Un pseudocódigo para el algoritmo sería:

¹Ver por ejemplo *Rod cutting* en el Cormen

Algoritmo 1 Programación dinámica *top-down* para calcular $f_c(i)$

$M \leftarrow$ vector de longitud n inicializado en \perp

procedure $F(i)$

if $i > n + 1$ o $i = n$ **then return** ∞

if $i = n + 1$ **then return** 0

if $M[i] \neq \perp$ **then return** $M[i]$

$M[i] \leftarrow \infty$

for $i + 1 \leq j < 2c$ **do**

$M[i] = \min(M[i], f(j + 1) + se_puede_armar_rima(i, j))$

return $M[i]$

Ejercicio 2

a) Dado $G = (V, E)$ un grafo conexo y $T_r = (V, E_T)$ un árbol DFS de G enraizado en $r \in V$ nos piden probar que r es punto de articulación de G si y solamente si r tiene grado mayor a 1 en T .

Probemos primero la vuelta. Sean v_1, \dots, v_k los hijos de r en T_r , con $k > 1$ por hipótesis (el grado de r es mayor a 1 en T_r). Como T_r es un árbol DFS los subárboles enraizados en los distintos v_i no tienen *cross edges* entre sí. Por lo tanto, todos los ejes de cada subárbol hacia afuera del subárbol van a r , y entonces al remover r estos subárboles quedan separados. Como $k > 1$ esto quiere decir que al remover r el grafo queda disconexo, y por ende r es un punto de articulación.

Para la ida, probemos el contrarrecíproco: supongamos que r tiene grado menor o igual a 1 en T_r , y veamos que entonces no es un punto de articulación. Si el grado de r es 0 entonces G es un único nodo, y por lo tanto r no es un punto de articulación (al eliminarlo no aumenta la cantidad de componentes conexas). Si el grado de r es igual a 1, notemos que dados dos nodos v, w del árbol distintos de r vale que el único camino simple de v a w en el árbol no pasa por r (debido al simple motivo de que cualquier camino simple que pasa por r tiene que terminar en r , ya que tiene grado 1). Es decir, todo par de nodos tiene un camino que los conecta sin pasar por r . Por lo tanto, al remover r esos caminos siguen estando, y se concluye que r no es un punto de articulación.

Vale la pena observar que la ida vale sin importar que el árbol sea DFS, mientras que en la vuelta usamos esa hipótesis fuertemente (cuando decimos que no hay *cross edges* entre los subárboles).

b) Tomemos un nodo v cualquiera que no sea raíz, y sean w_1, \dots, w_k sus hijos en el árbol T_r . Sabemos que los subárboles enraizados en los w_i no tienen *cross edges* entre sí. Por lo tanto, la única forma en que estos queden conectados si se elimina a v del grafo es que todos tengan ejes hacia nodos por arriba de v .

O sea, si v no es punto de articulación entonces todos estos subárboles tienen algún eje hacia un ancestro de v . Para la otra dirección, notemos que si todos estos subárboles tienen algún eje hacia un ancestro de v entonces al remover v los distintos subárboles siguen conectados con el resto del árbol (usando el eje que los conecta con un ancestro de v), y por lo tanto también están conectados entre sí. Luego, definimos el siguiente criterio:

Un nodo interno v del árbol T_r es punto de articulación si y solamente si alguno de los subárboles enraizados en sus hijos no tiene ejes hacia nodos que son ancestros de v

c) Dado un árbol DFS T_r de G (que se puede encontrar en tiempo lineal representando el grafo G como lista de adyacencias) es sencillo decidir si su raíz es punto de articulación en tiempo $O(1)$ usando el inciso a), así que solo nos falta ver como “implementar” el inciso b) en tiempo lineal.

Escribamos como $depth(v)$ a la profundidad del nodo v en el árbol T_r (es decir, $depth(r) = 0$, y los hijos de r tienen profundidad 1). Para resolver este problema nos gustaría saber, para cada subárbol, la menor profundidad a la que puede acceder tomando alguna *back edge*. Para esto, notemos como $low(w)$ a la menor profundidad que se puede alcanzar empleando alguna *back edge* del subárbol enraizado en w .

Supongamos que logramos calcular estos valores $depth(\cdot)$ y $low(\cdot)$ para todos los nodos, y que tenemos estos valores guardados en un vector. Luego, podemos decidir si un nodo cualquiera v es punto de articulación comparando $depth(v)$ con $low(w_i)$ para cada uno de sus hijos w_1, \dots, w_k . Más precisamente, v será punto de articulación si para alguno de sus hijos w_i vale que $depth(v) \geq low(w_i)$ (i.e. el subárbol enraizado en w_i no tiene ninguna *back edge* hacia nodos ancestros de v). Para checkear esto tenemos que hacer, por cada nodo v , a lo sumo $O(d(v))$ operaciones (usando como representación lista de adyacencias). Por lo tanto, podemos checkear todos los nodos en $O(\sum_{v \in V} d(v)) = O(m)$, que es lineal en el tamaño de entrada.

Entonces solo nos falta explicar cómo calcular estos vectores con los valores $depth(\cdot)$ y $low(\cdot)$ dado el árbol T_r . Los valores de $depth(\cdot)$ ya los calcula de por sí el DFS, por lo que solo falta ver cómo calcular $low(\cdot)$ en tiempo lineal. Observemos que vale

$$low(v) = \min\left(\min_{vz \in BE(v, T_r)} depth(z), \min_{w \in H(v, T_r)} low(w)\right) \quad (1)$$

donde $H(v, T_r)$ es el conjunto de hijos de v en el árbol T_r , y $BE(v, T_r)$ es el conjunto de *back edges* de T_r que inciden en v . Es decir, la menor profundidad que se puede alcanzar usando una *back edge* del árbol enraizado en v se alcanza usando alguna *back edge* que incide en v (lado izquierdo del mínimo), o bien usando alguna que incide en algún nodo en uno de los subárboles de sus hijos (lado derecho del mínimo).

²Y, técnicamente, tales que $depth(z) < depth(v)$

Esta relación recursiva ya nos da un algoritmo para calcular los valores de $low(\cdot)$. Para hacerlo de forma eficiente podemos usar programación dinámica sobre un vector de n posiciones: en ese caso, el costo de calcular todos los valores se puede computar como la sumatoria del costo de calcular cada valor, asumiendo que los llamados recursivos se resuelven en tiempo constante. Dado un nodo v cualquiera calcular $low(v)$ usando la ecuación 1 requiere recorrer todos los ejes que inciden en él, lo cual toma $O(d(v))$ (de nuevo, usando como representación lista de adyacencias). Por lo tanto, la complejidad final para resolver la recursión es $O(\sum_{v \in V} d(v)) = O(m)$. También deberíamos agregar un $O(n)$ para iniciar el vector donde se guardan los valores de $low(\cdot)$.

Resumiendo, nuestro algoritmo completo es el siguiente:

1. Dado G como lista de aristas, construimos una nueva representación de G como lista de adyacencias³. $O(n + m)$
2. Armamos un árbol DFS T de G , obteniendo en particular los valores de $depth(\cdot)$. $O(n + m)$
3. Calculamos con programación dinámica los valores $low(\cdot)$ sobre T . $O(n + m)$
4. Recorremos los nodos y para cada uno decidimos si es o no punto de articulación, usando el inciso a) o bien el b), de acuerdo a si es la raíz o no. $O(n + m)$

En base a los incisos a) y b) sabemos que este algoritmo es correcto, y aparte tiene la complejidad pedida.

d) Dado $G = (V, E)$, llamemos H al grafo que se obtiene removiendo todos los puntos de articulación de G . Si dos nodos v, w están conectados por un camino simple que no usa puntos de articulación entonces pertenecen a la misma componente conexa de H (ese mismo camino atestigua este hecho). De la misma forma, si dos nodos v, w no están conectados en H entonces no estaban conectados en G , o bien lo estaban pero todos los caminos que los unían pasaban por algún punto de articulación de G que no está en H . Por lo tanto, responder a la pregunta “¿Existe un camino simple entre v y w que no pase por puntos de articulación en G ?” es equivalente a la responder “¿Pertenecen v y w a la misma componente conexa de H ?”⁴.

Para responder a esta pregunta en $O(1)$ alcanza con tener un vector $componente(\cdot)$ que indique, para cada nodo v , a qué componente de H pertenece (por supuesto, habiendo numerado de alguna forma las componentes de H). Por lo tanto, la consulta

³En general se asume que los grafos de entrada siempre vienen representados como lista de aristas

⁴En realidad, las preguntas son equivalentes siempre y cuando v y w no sean puntos de articulación de G . En esos casos la respuesta es trivialmente **no**, ya que si v es punto de articulación entonces cualquier camino que una a v con otro nodo contendrá un punto de articulación (el nodo v)

de “¿Pertenece v y w a la misma componente conexa de H ?” se responde como $componente(v) == componente(w)$ con dos accesos $O(1)$. Este vector $componente(\cdot)$ se puede calcular fácilmente reconociendo las componentes conexas de H usando DFS o BFS.

Por lo tanto, el algoritmo que construye la estructura de datos para responder las queries consiste en:

1. Calcular el conjunto P de puntos de articulación de G , usando el algoritmo del inciso c). $O(n + m)$
2. Calcular H removiendo de G los nodos P . $O(n + m)$
3. Detectar las componentes conexas de H con DFS y preparar el vector $componente(\cdot)$. $O(n + m)$

Entonces, por cada query (v, w) respondemos **si** si $componente(v)$ es igual a $componente(w)$, y **no** en caso contrario. Esto toma $O(1)$ por query.

Ejercicio 3

a) Sea T un AGM de G y e el eje de menor peso de $G \setminus T$ (donde \setminus representa eliminar las aristas de T en G), queremos ver que es posible obtener un AG de segundo costo mínimo de G intercambiando e por alguna arista de T .

Sabemos por la teórica que si agregamos una arista de $G \setminus T$ a T se forma un ciclo simple, y que sacando cualquiera de las aristas de ese ciclo obtenemos un AG nuevamente. Tomemos entonces de las aristas del ciclo simple de $T + e$ la de mayor costo (sin contar a e), llamémosla e' . Llamemos también T' a $T + e \setminus e'$, y veamos que T' es un AG de segundo costo mínimo.

Observemos primero que $w(e) < w(e')$ porque si no $w(T')$ sería menor a $w(T)$, y T es un AGM (no pueden ser iguales por hipótesis del enunciado y por uno de los ejercicios de la práctica). Con esto, solo resta ver que no existe T'' tal que $w(T) < w(T'') < w(T')$.

Para esto, supongamos que T'' se obtiene agregando a T una arista $f \in G \setminus T$ (con $f \neq e$) y sacando una arista f' del ciclo que se forma. Debe ser entonces $w(f) > w(e)$ porque e era la de menor costo en $G \setminus T$. Notar, igual que con T' , que $w(f') < w(f)$ porque T'' no puede tener menor peso que un AGM. Veamos ahora que $w(T'') > w(T')$.

$$\begin{aligned}
 w(T'') &> w(T') \\
 w(T) + w(f) - w(f') &> w(T) + w(e) - w(e') \\
 w(f) - w(f') &> w(e) - w(e') \\
 w(f) &> w(e) - w(e') + w(f')
 \end{aligned}$$

Ahora siguiendo la sugerencia, $w(e) - w(e') + w(f') < w(e) + w(e') + w(f') < w(f)$, donde la primera desigualdad sale de que los pesos son positivos y la segunda de que f es una potencia de dos mayor que e , e' y f' y por lo tanto es mayor que la suma de todas ellas. Con esto queda probado que $w(T'') > w(T')$, y por lo tanto T' es un AG de segundo costo mínimo.

A modo de comentario, si nos ponemos quisquillosos, también habría que probar que las únicas alternativas relevantes son los T'' que se obtienen con un único intercambio de aristas, y no un AG cualquiera. Si bien en el parcial se consideró suficiente la justificación anterior, por completitud dejaremos la demostración del siguiente lema, del cual se deduce este hecho: sea T un AGM y T' un AG distinto de T . Luego, existe otro AG T'' con $w(T) \leq w(T'') \leq w(T')$ que difiere de T en una arista menos. La prueba del lema repite algunos de los razonamientos anteriores, aunque es un poco más técnica.

Recordemos que T y T' comparten algunas aristas, y otras no. Llamemos e a la *arista de menor costo de T' que no está en T* . Si le agregamos esta arista e a T sabemos que se forma un ciclo C . Aparte, todas las aristas en este ciclo tienen costo menor o igual al costo de e , ya que T es un AGM. Ahora, sabemos que alguna arista de C **no está** en T' . Esto vale porque, caso contrario, T' tendría un ciclo. Llamemos f a alguna de estas aristas, y recordemos que vale que $w(f) \leq w(e)$ por lo que dijimos en la mitad de este párrafo.

Vamos a agregar esta arista f a T' . Esto va a formar un nuevo ciclo C' , pero esta vez en T' . Notemos que *alguna arista de C' no está en T* (como antes, esto se deduce de que caso contrario habría un ciclo en T). Sea e' alguna de estas aristas. Vamos a ver que $w(f) \leq w(e')$.

e' es una arista que está en T' pero no en T . Eso quiere decir que, por como elegimos e , vale que $w(e) \leq w(e')$ (recordemos que e era la arista de menor costo entre aquellas que no estaban en T pero sí en T'). Aparte, por como elegimos f también sabemos que $w(f) \leq w(e)$, y de ambas desigualdades se deduce que $w(f) \leq w(e')$.

Con todo esto podemos definir el árbol T'' como $T'' = T' + f \setminus e'$. Esto es un árbol, dado que le estamos agregando una arista a T' (f era una arista que no estaba en T') y e' es una de las aristas que pertenece al ciclo que se forma al agregar f a T' . Aparte, notemos que

$$w(T'') = w(T') + w(f) - w(e') \leq w(T')$$

debido a que $w(f) - w(e') \leq 0$. Este nuevo árbol T'' es el que buscamos, dado que satisface $w(T) \leq w(T'') \leq w(T')$ y difiere en una arista menos de T (la f que le agregamos estaba en T , y la e' que sacamos no).

b) Dado el inciso **a)**, la idea sería calcular el AGM T , tomar la arista e de menor costo de $G \setminus T$ y reemplazarla por la de mayor costo del ciclo que se forma al agregarla a

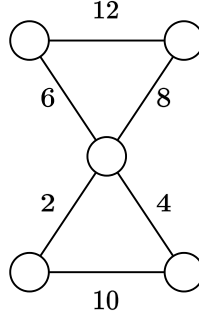


Figura 1: Contraejemplo para **3)c)**. El AGM de este grafo se obtiene eliminando las aristas de peso 10 y 12. El algoritmo de *b)* agregaría la arista de peso 10 y sacaría la de peso 4, obteniendo un *AG* de costo 6 mayor que el AGM. Sin embargo, si se agregase la de peso 12 y se sacase la de peso 8, se obtendría un *AG* de costo 4 mayor que el óptimo, con lo cual el *AG* elegido por el algoritmo no tiene el segundo costo mínimo.

T . T se puede calcular en $O(n^2)$ usando cualquier implementación de AGM para densos. Para obtener e en $O(n^2)$ podemos por ejemplo marcar en la matriz de adyacencia las aristas que no están en T y tomar la de menor costo de esas. Luego, para encontrar el ciclo en $T + e$, si $e = (v, w)$, podemos tomar el único camino en T entre v y w y esas son las aristas del ciclo. Para obtener ese camino podemos hacer BFS o DFS sobre T , en $O(n)$.

c) Notar que en la demostración del inciso **a)** usamos fuertemente el hecho de que las aristas son potencias de 2. En particular, si no fueran potencias de 2 podría pasar que $w(f) - w(f') < w(e) - w(e')$. Un ejemplo en el que pasa esto es el que se muestra en la figura 1.

d) En este inciso el objetivo es swapear los pesos de dos aristas de G obteniendo un grafo G' en el que nuestro algoritmo da el resultado correcto (por más que G tenga pesos pares y no potencias de dos). La idea para hacer esto es asegurarnos de que la diferencia entre el peso de la arista que agregamos y la que sacamos sea lo más chica posible. Como siempre se saca una arista del AGM y se agrega una de $G \setminus T$, queremos asegurarnos de sacar la más grande del AGM. Luego, como la que agregamos es la más chica de $G \setminus T$, si logramos esto podemos obtener la menor diferencia posible.

Si swapeamos el peso de la arista que sacaría nuestro algoritmo por el peso de la mayor arista del AGM, logramos exactamente esto. Notar que como es un swap de pesos entre dos aristas que están en el AGM, el AGM sigue siendo el mismo. Además, nuestro algoritmo elige la misma arista de $G \setminus T$ y la intercambia por la misma arista (solo que ahora esta última tiene peso mayor).